DISPENSA SVIGRUPPO

Git:

* ~:
  + [branchName/HEAD]~[numberN] :: simbolo che permette risalire il primo parent del commit puntato per N-generazioni (ex. HEAD~0 indica sé stesso, HEAD~1 indica il parent se history lineare oppure il primo parent da sinistra nel caso sia presente un merge prima del commit corrente)
* ^:
  + [branchName/HEAD]^[numberN] :: simbolo che permette di risalire il parent del commit puntato per N-generazioni (ex. HEAD^0 indica sé stesso, HEAD^1 indica il parent se history lineare oppure il primo parent da sinistra nel caso sia presente un merge prima del commit corrente, HEAD^2 indica il parent del parent se history lineare oppure il primo parent da sinistra del parent nel caso sia presente un merge prima del commit del parent oppure il secondo parent da sinistra nel caso sia presente un merge prima del commit corrente)

NB ~ e ^ si possono combinare per risalire l’albero dei parent in modi diversi!

* #!/bin/sh:
  + #!/bin/sh :: inserire questo codice a inizio dello script per farlo funzionare correttamente con Linux e salvare il file contenente lo script come file .sh (ad esempio “EsameGit.sh”) per indicare che è un file eseguibile da shell
* chmod:
  + chmod +x [nomeScript] :: esegue lo script fornito (per ricostruire la repo)
* cp:
  + cp [fileToCopy] [listOfFilesToCopyIn] :: copia il file indicato nei file indicati in seguito, sovrascrivendoli o creandoli se non presenti
* git add:
  + git add [fileName/directoryName] :: aggiunge all’index i file/directory indicate
* git branch:
  + git branch :: mostra tutti i branch presenti
  + git branch [branchName] :: crea un branch con nome branchName
  + git branch -d [branchName] :: elimina il branch indicato da branchName
* git cat-file:
  + git cat-file -p [hashCommit/hashTree/hashFile] :: mostra il contenuto del commit/tree/blob di cui gli viene passato l’hash

NB se un commit non ha parent vuol dire che è stato il primo commit effettuato!

NB se un commit ha due parent il primo dall’alto indica il branch su cui è stato fatto il merge!

* git cherry-pick:
  + git cherry-pick [hashCommit/branchName/tagName] :: prende il commit indicato e lo aggiunge come nuovo commit al branch corrente
  + git cherry-pick --abort :: interrompi cherry-pick
* git commit:
  + git commit -m [message] :: crea un nuovo commit con il messaggio indicato da aggiungere sul branch corrent
  + git commit --amend -m [message] :: crea un nuovo commit e lo sostituisce all’ultimo commit sul branch corrente, rendendo il commit sostituito dangling
  + git commit --amend --no-edit :: crea un nuovo commit e lo sostituisce all’ultimo commit sul branch corrente, rendendo il commit sostituito dangling ma mantenendo lo stesso messsaggio del commit sostituito sul nuovo commit

NB gli hash dei commit già presenti non vanno presi in considerazione dato che vengono calcolati tenendo conto di data e autore (per verificare correttezza repository vanno considerati solo hash dei tree ed hash dei blob)

* git fsck:
  + git fsck :: restituisce tutti i dangling blob e i dangling commit

NB per ottenere un dangling blob basta creare un file, aggiungerlo all’index e poi effettuare git reset, mentre nel caso di un dangling commit serve fare un commit e poi un reset a un commit precedente!

* git hash-object:
  + git hash-object [fileName] :: restituisce l’hash del file indicato
* git log:
  + git log :: mostra tutti i commit del branch corrente
  + git log [branchName] :: fa vedere i commit del branch indicato
  + git log --all --decorate --oneline --graph :: mostra sintesi repository come albero
  + git log --all --decorate --graph :: mostra repository estesa come albero
  + git log --all --graph :: mostra repository estesa come albero
  + git log --all --decorate --graph --date-order :: mostra repository estesa come albero con commit ordinati in base alla data in cui sono stati effettuati

NB se un commit non ha parent vuol dire che è stato il primo commit effettuato!

* git merge:
  + git merge [branchName] -m [commitMessage] :: effettua merge sul branch corrente tra ultimo commit del branch corrente e l’ultimo commit del branch indicato
  + git merge [tagName] -m [commitMessage] :: effettua merge sul branch corrente tra ultimo commit del branch corrent e il commit individuato dal tag sul nuovo commit
  + git merge --no-ff [branchName/tagName] -m [commitMessage] :: effettua merge sul branch corrente tra ultimo commit del branch corrente e l’ultimo commit del branch indicato o il commit individuato dal tag, dove --no-ff garantisce che venga creato un commit nuovo!
  + git merge --squash [branchName/tagName] :: aggiorna working directory e index del branch corrente come se fosse avvenuto il merge tra l’ultimo commit del branch corrente e l’ultimo commit del branch indicato o il commit individuato dal tag, senza tuttavia effettuare né il commit né il merge tra i branches

NB non è detto che una merge crei un nuovo commit, bensì questa proprietà deve essere esplicitato nei parametri: se infatti i due branch hanno stessa history (ovvero hanno i commit precedenti in comune) l’operazione di merge non farà altro che far avanzare il branch arretrato fino all’ultimo commit del branch più avanzato senza tuttavia creare un nuovo commit!

* git rebase:
  + git rebase [branchName] :: effettua il rebase del branch corrente spostandone il primo commit in seguito all’ultimo commit del branch indicato come parametro
  + git rebase --onto [newBaseBranch] [oldBaseBranch] :: effettua il rebase del branch indicato come secondo parametro spostandone il primo commit in seguito all’ultimo commit del branch indicato come primo parametro
* git revert:
  + git revert :: crea un nuovo commit in cui annulla i cambiamenti effettuati nell’ultimo commit
  + gir revert -n :: annulla i cambiamenti effettuati nell’ultimo commit senza creare un nuovo commit ma ripristinando semplicemente working directory e index allo stato precedente l’ultimo commit
  + git revert [commit] :: crea un nuovo commit che annulla tutte le modifiche dei commit effettuati a partire dal commit fornito come parametro incluso e quindi è come se aggiungesse in cima al branch corrente una copia del commit precedente al commit fornito
* git reset:
  + git reset [commit] :: ritorna al commit indicato e resetta solo l’index allo stato del commit indicato
  + git reset --soft [commit] :: ritorna al commit indicato ma non resetta nè working directory nè index
  + git reset --hard [commit] :: ritorna al commit indicato e resetta sia working directory sia index allo stato del commit indicato

NB per navigare tra i commit conviene sempre usare ~ e ^

* git rm:
  + git rm [fileName] :: rimuove il file indicato da index e working directory
  + git rm --cached [fileName] :: rimuove il file indicato solo dall’index
  + git rm -f [fileName] :: forza la rimozione del file indicato
  + git rm -rf [folderName] :: forza la rimozione ricorsiva di una folder e di tutti i file contenuti nella folder
* git stash:
  + git stash :: salva index e working directory in una pila LIFO
  + git stash pop :: ripristina index e working directory da pila LIFO
* git status:
  + git status :: mostra lo stato corrente di working directory e index
* git switch:
  + git switch [branchName] :: sposta la testa sul branch indicato
  + git switch -d [tagName] :: fai puntare la HEAD al commit con il tag indicato

NB per redigere lo script di creazione della repository se si deve spostare la HEAD su commit non puntati da branch bisogna usare i tag invece degli hash, dato che quest’ultimi vengono calcolati diversamente in base a ora e autore!

* git tag:
  + git tag [tagName] :: applica il tag tagName al commit più recente sul branch corrente
  + git tag -d [tagName] :: elimina il tag indicato
* mkdir:
  + mkdir [folderName] :: crea una cartella vuota
* printf:
  + printf [string] > [fileName] :: scrive la stringa nel file indicato, sovrascrivendolo o creandolo se non esiste
  + pritnf [string] >> [fileName] :: appenda la stringa nel file indicato, creandolo se non esiste

Eiffel:

* Parole riservate:
  + across loop :: ciclo simile al for loop composto dalle 3 parole *across, as* e *all/some* (across [iterable] as [renamedIterable] all/some [condizioni] end) usato per testare la veridicità di una o più condizioni sugli elementi di una struttura dati iterabile. Con *all* restituisce True se e solo se tutti gli elementi rispettano le condizioni indicate, invece con *some* restituisce True se e solo se almeno un elemento rispetta le condizioni indicate
  + and :: clausola AND
  + ANY :: è la classe root di Eiffel, come la classe Object in Java
  + attached :: serve per controllare che l’oggetto riferito non sia void (ex: *if attached x as x\_is\_*attached *then x\_is\_attached [istruzioni] end*)
  + create :: clausola a cui segue la dichiarazione dei costruttori da implementare nelle features successive
  + Current :: referenzia l’oggetto corrente e coincide con il this di Java
  + deferred :: significa che la classe/metodo è astratta/o (come abstract in Java)
  + detachable :: significa che l’oggetto/variabile indicata in seguito può essere Void
  + end :: serve per concludere ogni dichiarazione di classe/override/feature/do/if-elseif-else/loop
  + ensure :: serve per definire le postcondizioni
  + ensure then :: mette in AND le postcondizioni della sopraclasse e le postcondizioni della classe corrente (le postcondizioni di una sottoclasse si possono solo rafforzare ma non indebolire: “require no more, promise no less”)
  + feature :: introduce la dichiarazione di attributi e metodi della classe
  + for loop :: ciclo for che in Eiffel è costuito dalle 3 parole *from, until* e *loop* (from [condizioni] until [condizioni] loop [istruzioni] end)
  + if/elseif/else :: istruzioni utilizzate per testare condizioni (if [condizioni] then [istruzioni] elseif [condizioni] then [istruzioni] else [istruzioni] end)
  + implies :: [condizione] implies [conseguenza] (non va if prima della condizione indicata e si usa di spesso nelle postcondizioni per determinare lo stato corrente dell’oggetto in base a quello che era lo stato iniziale)
  + inherit :: coincide con l’ “extends” di Java ed è seguito dalla dichiarazione delle classi ereditate
  + invariant :: clausola a seguito della quale vengono definiti gli invarianti di classe, ovvero le proprietà che devono essere valide in ogni stato dell’oggetto e che vengono ereditate da tutte le eventuali sottoclassi e aggiunte in AND alle ulteriori invarianti dichiarate da quest’ultime (gli invarianti di una sottoclasse si possono solo rafforzare ma non indebolire: “require no more, promise no less”)
  + local :: clausola sotto cui dichiarare variabili locali
  + not :: clausola NOT
  + old :: parola riservata utilizzata nelle postcondizioni per indicare uno stato o una proprietà dell’oggetto precedente alla chiamate della funzione
  + or :: clausola OR
  + redefine :: dichiara i metodi di cui si effettuerà l’override nelle features e quindi per aggiungere precondizioni e postcondizioni nelle features ridefinite bisognerà utilizzare “require else” ed “ensure then”
  + require :: serve per definire le precondizioni
  + require else :: mette in OR le precondizioni della sopraclasse e le precondizioni della classe corrente (le precondizioni di una sottoclasse si possono solo indebolire ma non rafforzare: “require no more, promise no less”)
  + rescue :: coincide con il catch di Java ed è usato per gestire eccezioni
  + Result :: indica il valore di ritorno
  + retry :: se presente è l’ultima delle istruzioni correttive della rescue e permette di provare a rieseguire il metodo una volta gestita l’eccezione
  + Void :: indica riferimento nullo e coincide con il null in Java
* Simboli:
  + := :: operatore di assegnamento
  + = :: operatore di confronto
  + /= :: not equal